# Atlas

## Cross-Platform C++ Bitcoin Wallet

### Philip Glazman

#### Spring 2018

# Table of Contents

1. Introduction
2. Installation Instructions
3. User Manual
4. Design
5. Insights and Challenges
6. Conclusion
7. Bibliography
8. Source Code
9. Test Cases

# Introduction

##### Glossary

* Bitcoin – Network protocol used to reach consensus on who owns bitcoins.
* bitcoin – The value transferred in the Bitcoin protocol.
* Satoshis – The lowest denominator of bitcoin. One satoshi is 1/100millionth of a bitcoin.
* BIP – Bitcoin Improvement Proposals (BIP) are approved or pending proposals to the Bitcoin protocol. Several BIPs provide a standard for how the protocol or nodes should behave. This project uses several BIP standards regarding how wallets ought to be implemented.
* Mining – The process by which the network reaches consensus and a transaction is confirmed.
* Script – The programming language used by Bitcoin to write scripts. This language operates uses operation codes on a reverse polish notation stack.
* Smart Contract – A piece of code that is self-enforcing on the blockchain.
* Blockchain – Public data structure that maintains a ledger representing the entire state of the network.

For more information on these terms and others not covered, please consider reviewing the open-source Bitcoin wiki located here: <https://en.bitcoin.it/>

##### Background

Bitcoin is a digital money developed in 2009 where each node participating in the network can independently validate transactions and propagate them throughout the network using software similar to bittorrent. The protocol relies on public-key cryptography to create public addresses for the end-user. In terms of bitcoin, a wallet software manages the private keys that are associated with each public address. These keys gives users ownership in spending transactions and bitcoin. In bitcoin, the wallet is an abstraction that allows the end-user to send and receive payments.

##### Project

The aim of this project is to create a user-friendly bitcoin wallet implementation that encourages self-ownership of bitcoins and the use of bitcoin’s Script language. Several Bitcoin wallets exist in the ecosystem but there does not exist a wallet that provides an abstraction layer that allows the end-user to interact with bitcoin smart contracts in a user-friendly way.

Atlas proposes a different way for the user to interact with how interact with Bitcoin. With a focus on financial independence through education, Atlas provides a straightforward way for the user to write smart contracts and learn more about the underlying low-level protocol.

##### Warning

This wallet was not extensively tested for security vulnerabilities, therefore should not be used with real bitcoin. The current implementation of Atlas operates on the Bitcoin test network and uses test network bitcoins. Funds sent and received should be used with addresses that have a test network prefix.

##### Other Notes

This project heavily relied on Andreas M. Antonopoulos’s *Mastering Bitcoin*, open-source documentation notes on Libbitcoin on the Libbitcoin Wiki, and Aaron Jaramillo’s tutorials on Libbitcoin. These, among other scattered documentation along the web, were very helpful and resourceful. Several illustrations are used and referenced in this documentation that are from *Mastering Bitcoin*.

# Installation Instructions

Before running Atlas, a couple of important libraries are needed on the local machine.

1. Boost

Visit <https://www.boost.org/users/download/>

$ brew install boost

1. Libbitcoin

Visit <https://github.com/libbitcoin/libbitcoin/tree/version3>.

$ ./autogen.sh

$ ./configure

$ make

$ sudo make install

$ sudo ldconfig

More details on Libbitcoin installation can be found on Github README.md

1. Curl

Visit <https://curl.haxx.se/download.html>

$ brew install curl

1. JsonCPP

Visit <https://github.com/open-source-parsers/jsoncpp>

$ brew install jsoncpp

1. OpenSSL

Visit <https://www.openssl.org/>

$ brew install openssl

1. Run make file in /qt.

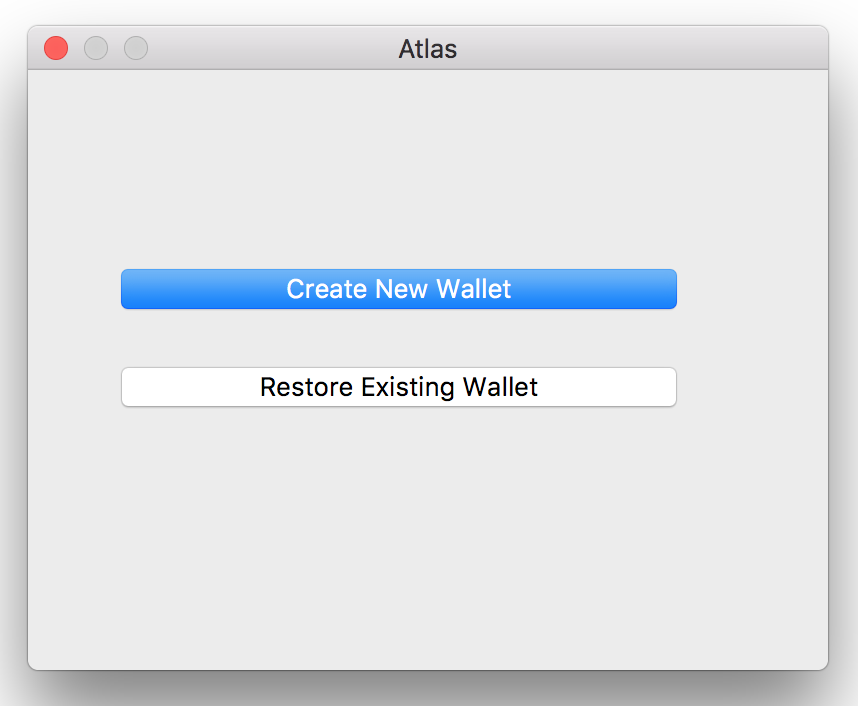
# User Manual

##### Warning

Atlas uses industry standards for wallet management but there are several risks involved. As noted in the warning section in the Introduction, this wallet should not be used with real bitcoin.

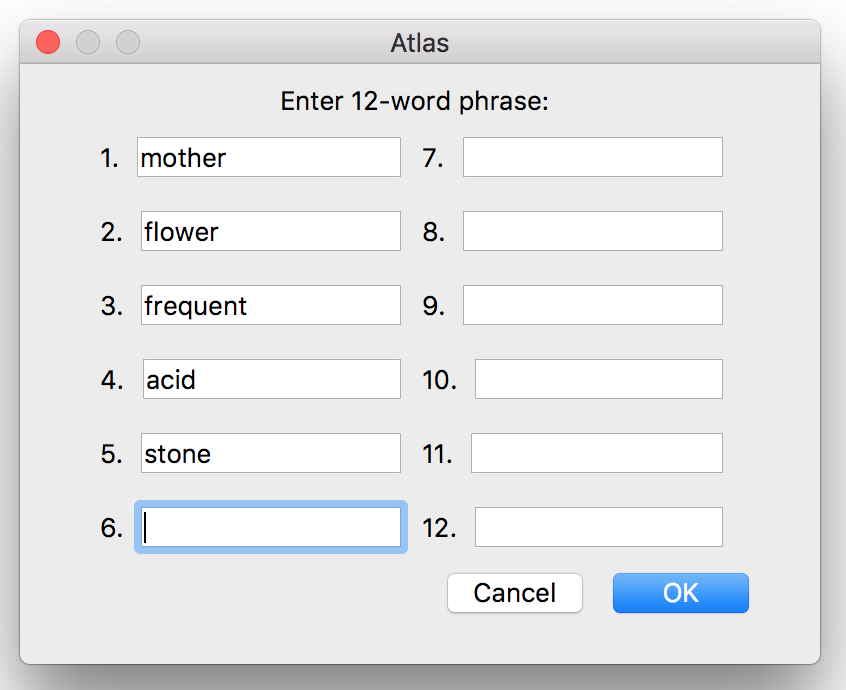
##### Wallet Creation

Upon starting the wallet, the user is prompted with a choice to either start a new wallet or restore an existing wallet. When a user starts a new wallet, a new seed is created that maintains the wallet. When a user chooses to restore a wallet, Atlas prompts the user for twelve words that comprises the mnemonic phrase. The user should keep these mnemonic words secret as they are the key to the wallet.



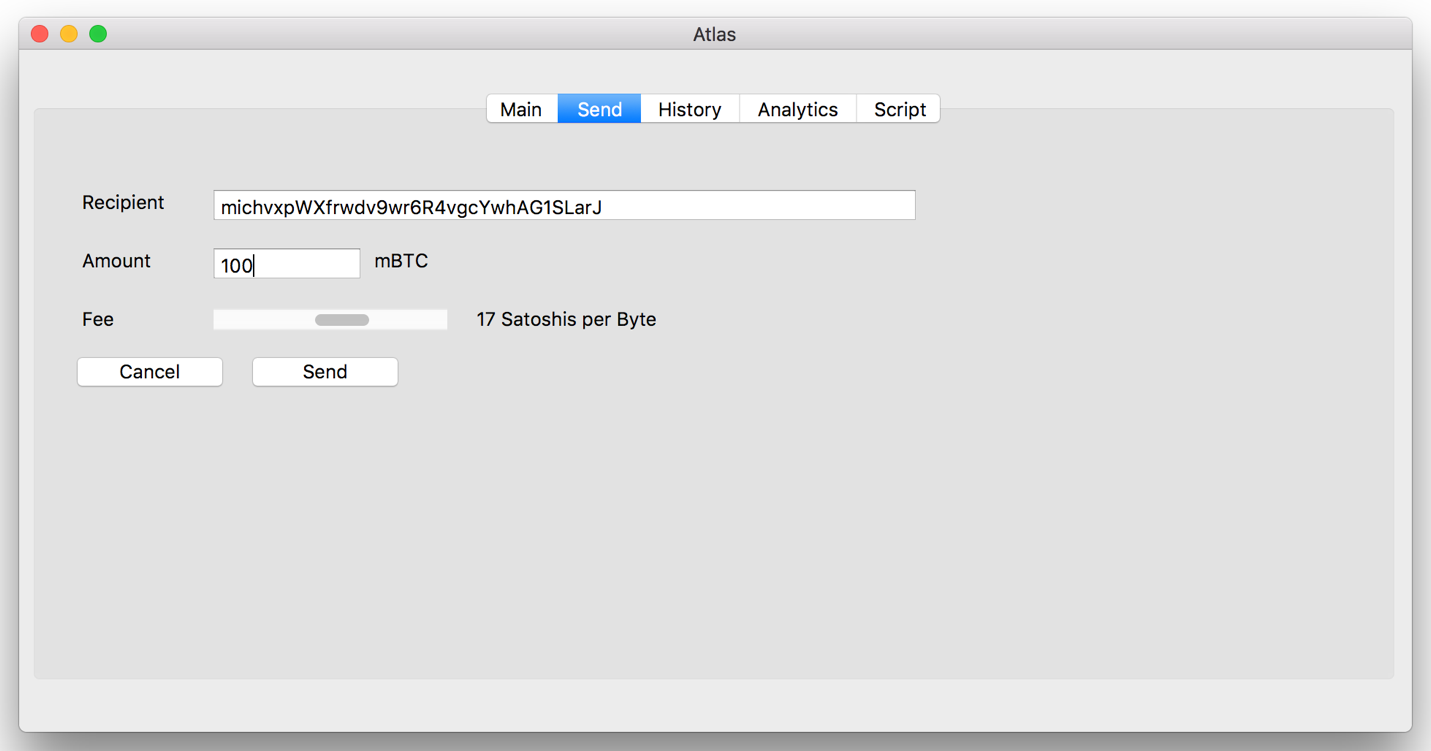
##### Mnemonic Phrase

The mnemonic phrase is a set of twelve words that create the wallet and make the wallet unique. This phrase should be kept secret for security. In its current implementation, the user cannot export their phrase. For development, a mnemonic phrase is included in the documentation in the Test Cases section. When starting the wallet, simply enter the 12 words into the boxes and it will restore an existing development wallet.



##### Sending Bitcoin

Sending a bitcoin transaction is very straightforward. A transaction in the Send tab allows the user to construct a basic transaction. A recipient represents another bitcoin payment address that the user will send funds to. The amount represents the number of bitcoins to send to the recipient. It is denominated in mBTC. The fees slider allows the user to change the amount of satoshis paid for the transaction fee. The user can have a minimum of zero satoshis per byte fees. It is important to note that Atlas sets a maximum fee which corresponds to the fee for the fastest transaction in the Analytics tab. This is implemented so that the user cannot overspend in fees.

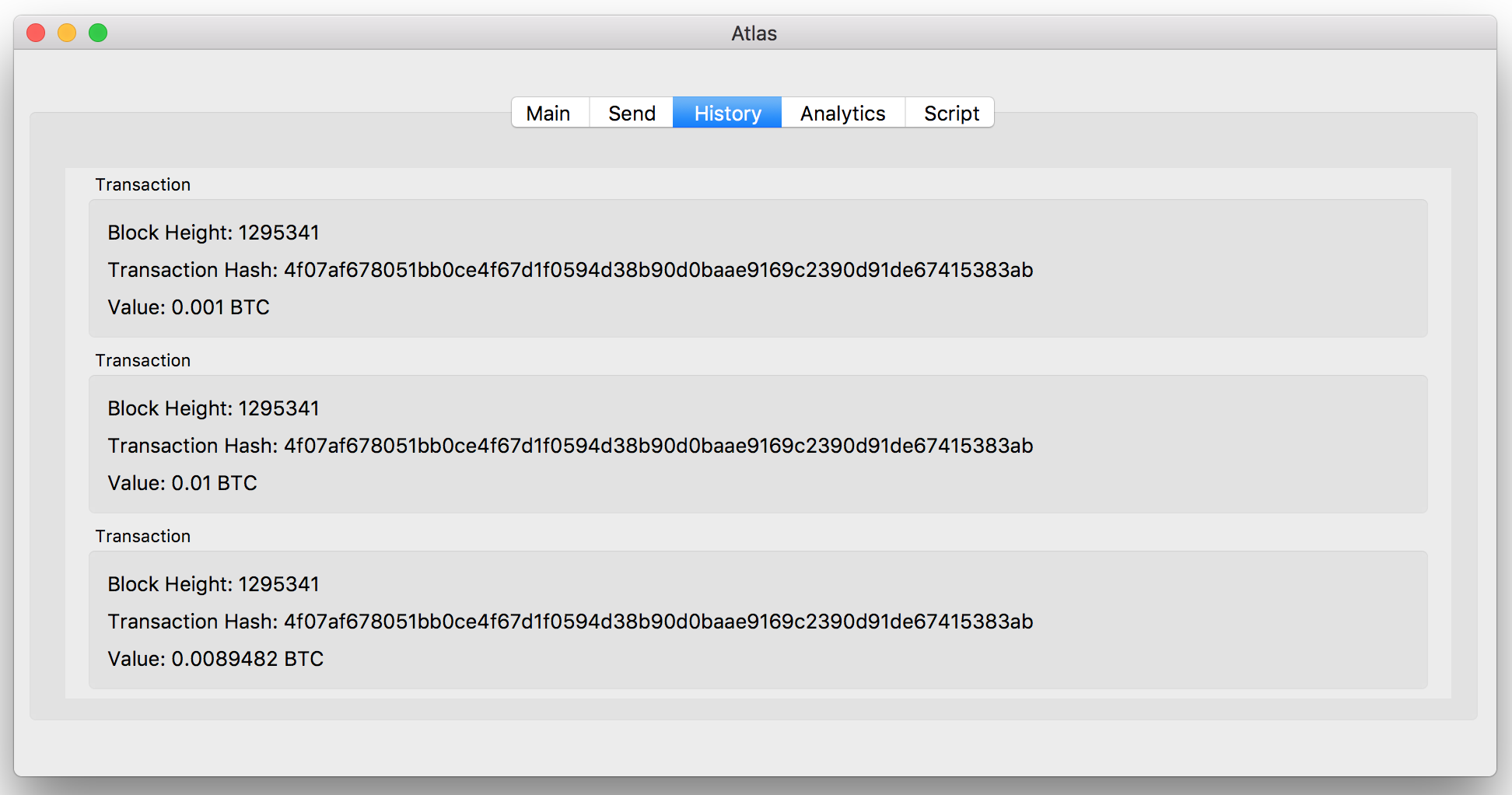


Below is a chart of bitcoin denominations for reference. It is from the Bitcoin wiki.



##### Transaction History

The History tab includes a list of transactions done by the addresses associated with the wallet. Each box in scroll area includes a transaction with a block height, transaction hash, and value. The block height represents where in the blockchain the transaction was confirmed, or mined. The transaction hash is a unique identifier for the transaction that the user can later reference. The value is the value of bitcoin transacted in that transaction.

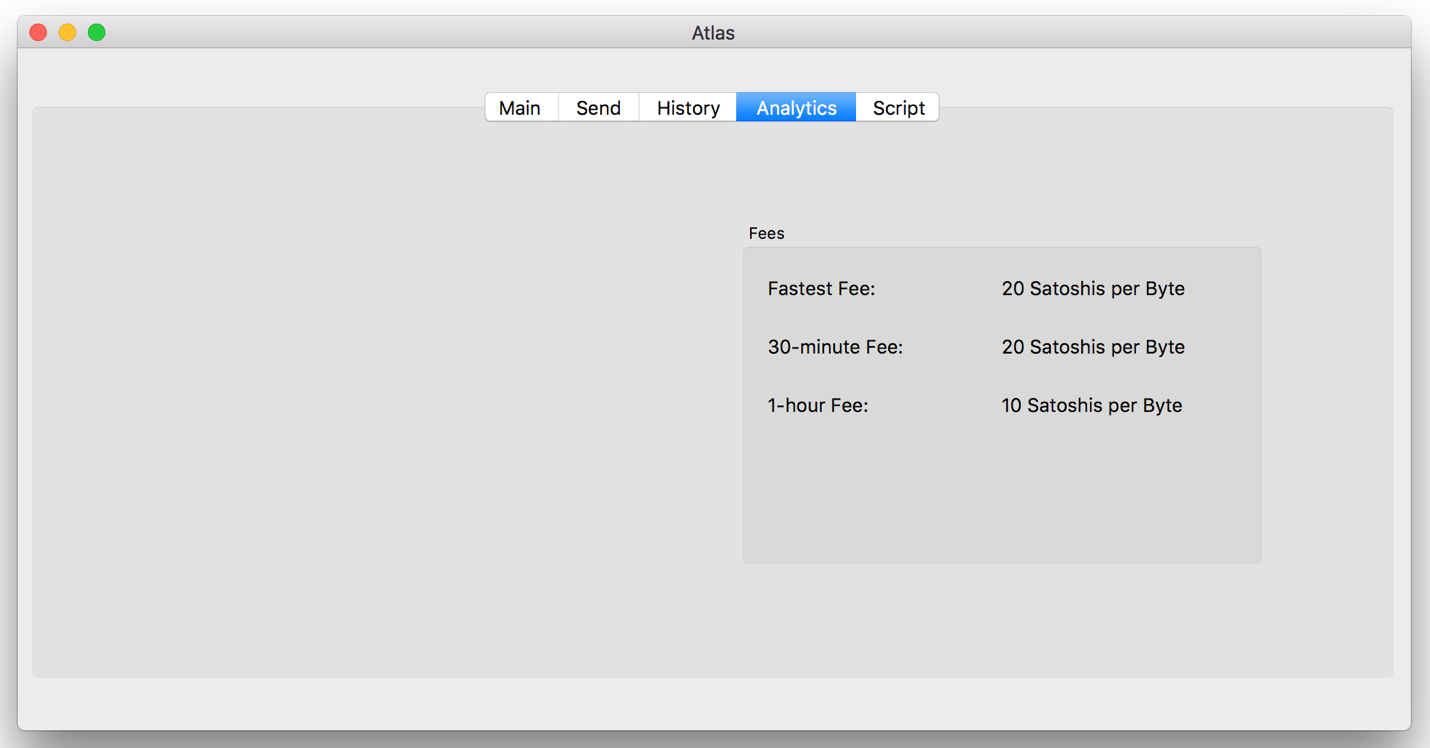


##### Understanding Network Fees

Transactions sent through the Bitcoin network may include a fee. The fee is optional and is set by the sender. A higher fee signals to the network that a specific transaction should be given priority for confirmation while a lower fee can lead to longer confirmation times. The fee market in Bitcoin is a free market set by supply and demand. As a result, the fee market changes over time and transactions can cost differently over the course of a day.

Atlas in its current implementation uses bitcoinfees.earn.com, a third-party API, to receive a suggested transaction fee. Using this API, Atlas is able to suggest to the user three fee costs located in the Analytics tab of Atlas. The user can choose to ignore these suggestions and selected a different fee.

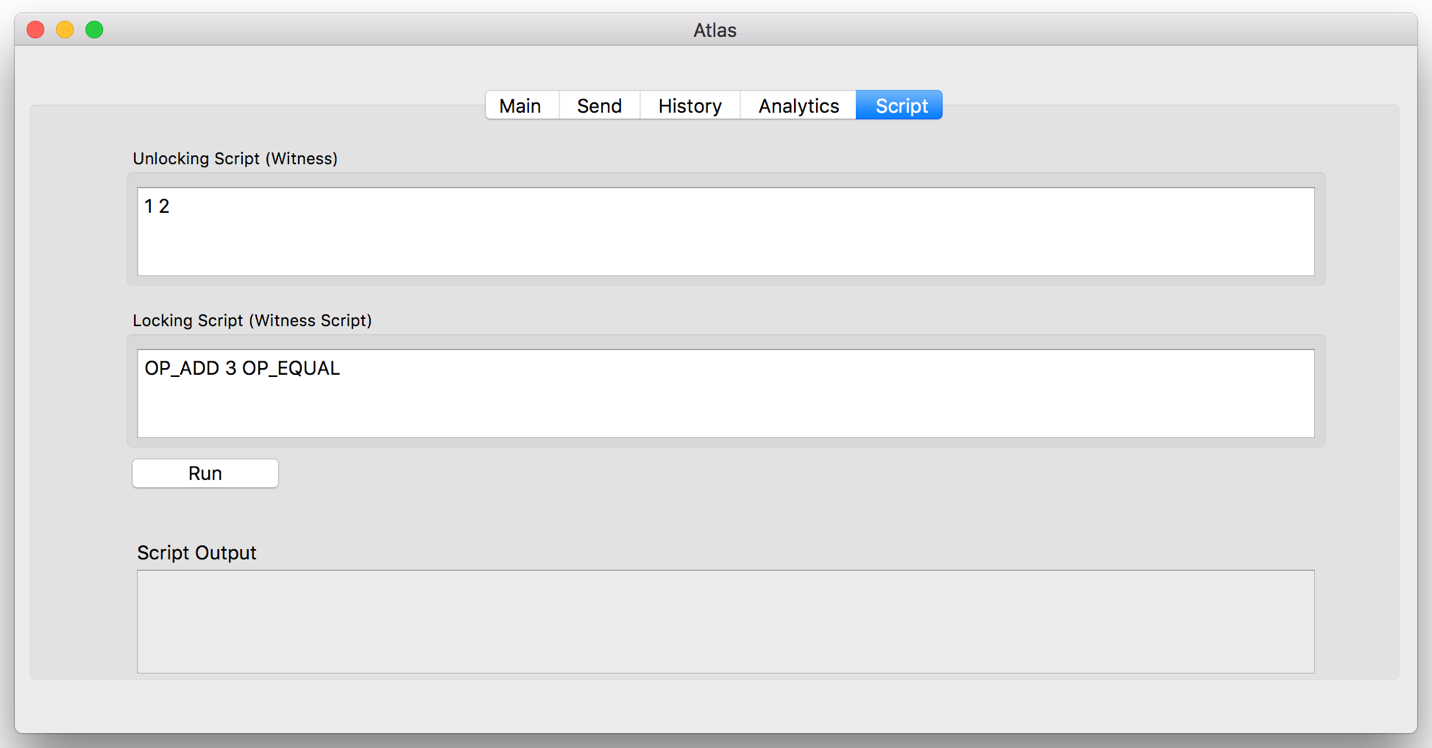
Fees are represented as Satoshi per Byte. This measurement informs the user that it costs n Satoshi per byte. If a transaction is 250 bytes in size and a suggested fee is 40 Satoshis per Byte, the fee will be 250\*40 Satoshis.



##### Using Bitcoin Script

Bitcoin protocol uses a stack-based programming language called Script that allows the user to create smart contracts and develop on the Bitcoin protocol. The Script tab in Atlas provides a basic interface for the user to learn and engage with Bitcoin Script by validating scripts. Witness and Witness Script serve as text edits where the user can write a basic script.

In this example, 1 and 2 are pushed onto the stack. Operation code OP\_ADD pops two items off the stack, adds them, and pushes the result onto the stack. Next, 3 is pushed onto the stack. Operation code OP\_EQUAL evaluates that the two top items are equal. Given that they are equal in this case, a True Boolean is pushed onto the stack and the stack executes successfully. With only a True Boolean left on the stack, the script is valid.



Below is a list of available operation codes that can be used in the Atlas script interpreter. The definitions for each operation code is from Antonopoulos’ *Mastering Bitcoin.*

|  |  |
| --- | --- |
| Available Operation Codes |  |
| OP\_DROP | Pop the top item in the stack. |
| OP\_DUP | Duplicate the top item in the stack. |
| OP\_DEPTH | Count the items on the stack and push the resulting count. |
| OP\_EQUAL | Push TRUE (1) if top two items are exactly equal, push FALSE (0) otherwise. |
| OP\_1ADD | Add 1 to the top item. |
| OP\_1SUB | Subtract 1 from the top item. |
| OP\_NEGATE | Flip the sign of the top item. |
| OP\_ABS | Change the sign of the top item to positive. |
| OP\_ADD | Pop top two items, add them and push result. |
| OP\_SUB | Pop top two items, subtract first from second, push result. |
| OP\_NUMEQUAL | Return TRUE if top two items are equal numbers. |
| OP\_NUMNOTEQUAL | Return TRUE if top two items are not equal numbers. |
| OP\_LESSTHAN | Return TRUE if second item is less than top item. |
| OP\_GREATERTHAN | Return TRUE if second item is greater than top item. |
| OP\_LESSTHANOREQUAL | Return TRUE if second item is less than or equal to top item. |
| OP\_GREATERTHANOREQUAL | Return TRUE if second item is greater than or equal to top item. |
| OP\_MIN | Return the smaller item of the two top items. |
| OP\_MAX | Return the larger of the two top items. |
| OP\_WITHIN | Return TRUE if the third item is between the second item (or equal) and first item. |
| OP\_RIPEMD160 | Return RIPEMD160 hash of top item. |
| OP\_SHA1 | Return SHA1 hash of top item. |
| OP\_SHA256 | Return SHA256 hash of top item. |
| OP\_HASH160 | Return RIPEMD160(SHA256(x)) hash of top item. |
| OP\_HASH256 | Return SHA256(SHA256(x)) hash of top item. |

The script output section of this tab serves as a console for the user where it is printed if a script is valid or not.

# Design

##### Philosophy

The wallet is designed in object-oriented principles. Objects are modularized and data is encapsulated in order to preserved object-oriented principles. The wallet is organized into basic wallet functionality, Bitcoin script manipulation, network functionality, utilities, and the front-end application.

##### Components

### List of high-level components that make the wallet fundamentals.

Base Wallet

|  |  |
| --- | --- |
| Components | Description |
| Mnemonic Code Words | Following a wallet standard, generated entropy will translate to 12 English words from a set. These words in addition to salt, will lead to a seed that creates a unique wallet. These 12 words could be written down and entered into the wallet to create this same unique wallet. |
| HD Wallet | A HD wallet, or deterministic wallet, is a wallet that creates a keychain based on a 512 bit seed. This is a standard in current Bitcoin wallets. |
| Bloom Filter | Bloom Filter is a standard privacy feature that allows the user to query for transactions without revealing to the network the specific transactions that he/she is asking for. |
| Peer Networking | Wallet connects to peers, does hand-shaking, and ask for transaction data. Most of the low-level work is handled by Libbitcoin library. |
| Payments | Allow user to send bitcoin and see the amount of bitcoin received. Allow user to generate new unique address for when receiving bitcoin |
| Graphical User Interface | Cross-platform Qt5 GUI. |

Analytics

|  |  |
| --- | --- |
| Components | Description |
| Fee Estimation | Query network to determine low, high, and median transaction fees. Provide recommendation to user for fee cost and when to send transaction. |
| Spend Analysis | Provide user with information on where bitcoin have been spent. |

Script

|  |  |
| --- | --- |
| Components | Description |
| Console | User can write their own bitcoin scripts and send them as transactions. Bitcoin Script language is stack-based language with limited OP codes. |
| Script debugger | Include debugger to help catch any errors in the user’s script. |

##### Classes

List of classes for backend and frontend of Atlas.

|  |  |
| --- | --- |
| Network Classes |  |
| Network | * Provides network functionality to the wallet including broadcasting transactions, reading data from Bitcoin blockchain, and accessing the bitcoinfees.earn.com API for transaction fee recommendations. * Utility functions for accessing fee recommendations. |
| Bloomfilter | * Privacy feature for querying network for inputs in a transaction including unspent transaction outputs (utxo). |

|  |  |
| --- | --- |
| Script |  |
| Script | * Provides functionality of Bitcoin script by simulating operations on a stack. * Includes a stack that serves as the execution stack for Bitcoin script. |
| Operation | * Includes operation codes and their functionality. Each operation code performs a function on the provided execution stack. * Includes utility cryptographic functions that can be used on the Bitcoin execution stack. |

|  |  |
| --- | --- |
| Utility |  |
| Valid\_Address | * Utility for validating if a string matches the consensus rules of the Bitcoin blockchain. |

|  |  |
| --- | --- |
| Wallet |  |
| Wallet | * Creates wallet seed. * Responsible for key management. * Responsible for address management. |
| utxo | * Manages the wallet’s record of unspent transaction outputs. |
| Transaction | * Provides utility functions for building a transaction. * Functionality to broadcast transaction when connected to the network. * Holds a history of previous transactions related to the wallet. |
| Error | * Handles exceptions and has an error stack. * Provides error log for debugging. |

|  |  |
| --- | --- |
| Qt |  |
| App | * User interface for the main wallet application. * User interface is divided into four tabs. |
| Restore\_wallet | * User interface for mnemonic prase input. * Validation of user input. |
| Start\_menu | * User interface for selecting to start a new wallet or restore an existing wallet. |

##### Data Structures

Important data structures that provides functionality for wallet fundamentals.

|  |  |
| --- | --- |
| Data Structures |  |
| feeEstimation | * Struct stores three fee recommendations to the user. * Contains satoshi value for fees that will result in transaction confirmation in fastest time, 30 minutes, or 60 minutes. |
| m\_ErrorMsgs | * Error stack that contains any errors that occurred within the wallet during runtime. |
| m\_tx | * Tuple data type representing any spent and confirmed transaction. * Contains satoshi value of transaction, transaction hash, and block height of the confirmed transaction. |
| m\_utxo | * Vector of tuples holding all unspent transaction outputs. These transactions outputs are spendable by the wallet. * Tuple contains satoshi value of transaction, transaction hash, and Bitcoin payment address. |

# Insights and Challenges

##### UTXO Management

The management of each transaction under the hood was more difficult than planned. The function of a wallet is to create an efficient manner for organizing transactions. Each payment address might have several different transactions associated with it. Each transaction will have varying values of bitcoins. The aim of any wallet is to provide a way to send bitcoin from any previous transaction. This becomes complicated fairly quickly as organizing transactions has a direct affect on the fees that the user pays. If an outgoing transaction relies on many input transactions, the fees paid by the user will be higher because the transaction size increases. Atlas uses a basic algorithm for sorting unspent transaction outputs on value. Going forward, a weighted approach will be needed so that value and number of inputs can be taken into account.

##### Transaction Building

Atlas currently only builds transactions that are pay-to-public-key-hash (P2PKH) which constitutes more than 80% of all transactions on the bitcoin network. The aim of Atlas was to experiment with building transactions that are more complex and less prevalent. There is still significant work to be done on transaction building as well as more efficient organization on how transactions are built.

# Conclusion

Writing this program was intensive ultimately I am still not satisfied with the product and will have to continue to update the project. There are several features that must be added in order for the wallet to become reliable and be able to use real Bitcoin funds. In this section, I will review the overall opportunities gained in designing the wallet and what I plan to add to it.

##### Opportunities

The opportunities in education gained by building a bitcoin wallet are very meaningful. The Bitcoin protocol has been in uninterrupted operation for nine years and does not show signs in losing relevancy. The venture capital money, developer interest, and philosophical intrigue into Bitcoin makes it worthwhile for at least some brief interest for any computer scientist.

The motivation in building Atlas were found in a desire to better understand the Bitcoin protocol. Most Bitcoin users interact with the protocol through the wallet abstraction layer, therefore understanding the mechanics of this software can allow a developer to significantly improve the way in which people interact with the protocol. Atlas was designed as an educational product that I hope to soon develop into an industry level product. The exciting challenges in learning how transactions and protocol work has also invited new ideas to explore going forward.

On a technical level, designing Atlas has made me more comfortable in designing large programs as well as have a better understanding on how to design object-oriented code. Atlas is not a perfect example of object-oriented principles, but I now know the underlying issues in order to challenge the code base and make it near-perfect.

The C++ language for this project for its object-oriented design and flexibility of memory management. The language is cross-platform which allows for flexibility on the devices that can run it. The Libbitcoin bitcoin development library was very resourceful in designing the wallet because it was able to abstract several low-level cryptography and functionality. Boost library was also resourceful in using property trees data structure. The C++ is time-tested, has significant developer resources, and is overall a very flexible language that gives the programmer a large amount of control over detail.

Using the Libbitcoin library API was very challenging initially due to the scarce resources available for its latest version. However, the documentation is growing and being updated more consistently since starting Atlas.

Bitcoin’s ecosystem is nascent, but there are several industry standards that were used in the design of Atlas. In particular, BIP 39 for mnemonic phrase and BIP 32 for the hierarchical design of the wallet’s keychain.

Designing Atlas required a large learning curve about Bitcoin and a basic understanding of its cryptography. Andreas Antonopoulos’ *Mastering Bitcoin* was incredibly resourceful that provided a strong technical foundation in Bitcoin’s protocol.

The graphical user interface of Atlas was written in Qt5 which is a cross-platform library for cross-platform applications. Qt5 has both extensive documentation and very reliable codebase for a free product. As part of using the library, it was agreed that Atlas will remain open-source.

##### Next Steps

There are several next steps to Atlas in order to improve the functionality and reliability of the application. In particular, BIP 21 will be implemented in order to allow QR codes to be presented. In addition, better fee recommendations and analytics will need to be added in order for the user to have a better understanding of their funds. In the initial proposal of the wallet, providing analytics was a major pillar of the application. Unfortunately, Atlas only provides fee estimation through a 3rd party API. Going forward, it must not rely on the API and instead use dynamic fee estimation. In addition to fee estimation, an analytics dashboard will be implemented that shows where and how bitcoin are spent.

There is still significant work to be done for the user’s exposure to the Bitcoin script language. In its current form, the user can only test and debug a Bitcoin script. Going forward, the user should be provided a way to submit transactions with their written script. In addition, after conversations with helpful developers as well as reading supplementary material, there are new ways in developing a more user-friendly approach to script construction that I will have to implement.

# Bibliography

The following resources were very helpful in building Atlas.

Antonopoulos, Andreas M. *Mastering Bitcoin: Programming the Open Blockchain.* 2nd ed., O'Reilly Media, 2017.

Jaramillo, Aaron. “The Libbitcoin Tutorials.” *The Web Log of Aaron Jaramillo*, aaronjaramillo.org/category/libbitcoindocs.

“Libbitcoin Wiki.” *GitHub, Github*, github.com/libbitcoin/libbitcoin/wiki.

“Libbitcoin Documentation.” *Overview - Libbitcoin 1 Documentation*, libbitcoin.dyne.org/doc/overview.html.

# Source Code

# Test Cases

Launch new wallet.

Restore existing wallet using mnemonic phrase:

|  |  |
| --- | --- |
| scatter | blanket |
| found | mother |
| issue | frequent |
| friend | acid |
| front | shaft |
| glare | loud |

1. Sending a transaction to the address located on the main tab.

2. Check if the transaction was successfully broadcasted using <https://live.blockcypher.com/btc-testnet/>

3. Test a simple bitcoin script:

Witness: 1 2

Witness Script: OP\_ADD 3 OP\_EQUAL

4. Compare the history of transactions in the history tab with the blockcypher explorer.